Graph-based keyword search in heterogeneous data sources

Angelos Christos Anadiotis[1], Mhd Yamen Haddad[2], Ioana Manolescu[2]
[1]Ecole Polytechnique and Institut Polytechnique de Paris, [2]Inria and Institut Polytechnique de Paris,
{name.surname} @ [1]polytechnique.edu, [2]inria.fr

ABSTRACT

Data journalism is the field of investigative journalism which focuses on digital data by treating them as first-class citizens. Following the trends in human activity, which leaves strong digital traces, data journalism becomes increasingly important. However, as the number and the diversity of data sources increase, heterogeneous data models with different structure, or even no structure at all, need to be considered in query answering.

Inspired by our collaboration with Le Monde, a leading French newspaper, we designed a novel query algorithm for exploiting such heterogeneous corpora through keyword search. We model our underlying data as graphs and, given a set of search terms, our algorithm finds links between them within and across the heterogeneous datasets included in the graph. We draw inspiration from prior work on keyword search in structured and unstructured data, which we extend with the data heterogeneity dimension, which makes the keyword search problem computationally harder. We implement our algorithm and we evaluate its performance using synthetic and real-world datasets.

1 INTRODUCTION

Data analysis is increasingly important for several organizations today, as it creates value by drawing meaningful insights from the data. As we are moving towards large data lakes installations where huge amounts of data are stored, the opportunities for important discoveries are growing; unfortunately, on par with the useless information. Moreover, the data to be processed is often stored in different formats, ranging from fully and semi-structured, to completely unstructured, like free text. Accordingly, the challenges in processing all this data that is available today, reside in both expressing and answering queries.

Research in heterogeneous data processing has proposed several approaches in addressing the above challenges. On the one side, massively parallel processing systems like Spark [33], Hive [28] and Pig [26] provide connectors for heterogeneous data sources and allow the execution of data analysis tasks on top of them, using either a platform-specific API or a query language like SQL. Polystore-based approaches [3, 9, 12] focus more on the data model and the query planning and optimization on top of heterogeneous data stores. Finally, the so-called just-in-time (JIT) data virtualization approach generates the query engine at runtime based on the data format [20, 21]. All these works consider that users, typically data scientists, already know what they are looking for, and they express it either using a powerful query language or a rich API.

However, today the data analysis paradigm has shifted and a central point is to find parts of the data which feature interesting patterns. The patterns may not be known at query time; instead, users may have to discover them through a process of trial and error. A popular query paradigm in such a context is *keyword search*. A staple of Information Retrieval in data with little or no structure, keyword search has been applied also on relational, XML or graph data, when users are unsure of the structure and would like the system to identify possible connections. In this work, we model a set of heterogeneous data sources as a graph, and focus on **answering queries asking for connections among the nodes of the graph which are of interest to the users**. This work is inspired from our collaboration with Les Décodeurs, Le Monde's fact-checking team[1], within the ContentCheck collaborative research project[2]. Our study is novel with respect to the state of the art (Section 6) as we are the first to consider that an answer may span over multiple datasets of different data models, with very different or even absent internal structure, e.g., text data. For instance, a national company registry is typically relational, contracts or political speeches are text, social media content typically comes as JSON documents, and open data is often encoded in RDF graphs.

Integrated graph preserving all original nodes In the data journalism context mentioned above, it is important to be able to *show where each piece of information in an answer came from*, and *how the connections were created*. This is a form of provenance, and can also be seen as result explanation. Therefore, the queried graph needs to preserve the identity of each node from the original sources. At the same time, to enable interesting connections, we: (*i*) extract several kinds of meaningful entities from all the data sources of all kinds; (*ii*) interconnect data sources that comprise the same entity, or very similar ones, through so-called *sameAs*. Both extraction and similarity produce results with some *confidence*, a value between 0 and 1, thus, some edges in our graph have can be seen as uncertain (but quite likely).

No help from a score function An important dimension of keyword search problems is *scoring*, i.e., how do we evaluate the interestingness of a given connection (or query result). This is important for two reasons. First, in many scenarios, the number of results is extremely large, users can only look at a small number of results, say k. Second, some answer score measures have properties that can help limit the search, by allowing to determine that some of the answers not explored yet would not make it into the top k. Unfortunately, while desirable from an algorithmic perspective (since they simplify the problem), such assumptions on the cost model are not always realistic from a user perspective, as we learned by exchanging with journalists; we detail this in Section 3.

Bidirectional search All edges in our graph are *directed*, e.g., from the subject to the object in an RDF graph, from the parent to the child in a hierarchical document etc., and, in keeping with our goal of integral source preservation, we store the edge direction in the graph. However, we allow answer trees to traverse edges in any direction, since heterogeneous data sources may model the

[1]http://www.lemonde.fr/les-decodeurs/
[2]https://team.inria.fr/cedar/contentcheck/

same information either, say, of the form Alice $\xrightarrow{\text{wrote}}$ Paper$_1$ or Paper$_1$ $\xrightarrow{\text{hasAuthor}}$ Alice; since users are unfamiliar with the data, they should not be penalized for not having "guessed" correctly the edge directions. This is in contrast with many prior works (see Section 6) which define answers as a tree where from the root, a node matching each keyword is reached by traversing edges in their original direction only. For instance, assume the graph comprises $a_1 \xrightarrow{\text{wrote}} p_1$ and $a_2 \xrightarrow{\text{wrote}} p_1$. With a restricted notion of answers, the query $\{a_1\ a_2\}$ has no answer; in contrast, in our approach, the answer connecting them through p_1 is easily found. Bidirectional search gives a functional advantage, but makes the search more challenging: in a graph of $|E|$ edges, the search space is multiplied by $2^{|E|}$.

The contributions made in this work are as follows:

- We formalize the problem of bidirectional keyword search on graphs as described above, built from a combination of data sources.
- With respect to scoring, we introduce a general score function that can be extended and customized to reflect all interesting properties of a given answer. We show that this generality, together with the possibility of confidence lower than 1.0 on some edges, does not enable search to take advantage of simplifying assumptions made in prior work.
- We propose a complete (if exhaustive) algorithm for solving the keyword search problem in this context, as well as some original pruning criteria arising specifically in the context of our graphs. Given the usually huge search space size, a practical use of this algorithm is to run it until a time-out and retain the best answers found.
- We have implemented our algorithm and present a set of experiments validating its practical interest.

A previous version of our system had been demonstrated in [5]. Since then, we have completely re-engineered the graph construction (this is described in the companion paper [4]), deepened our analysis of the query problem, and proposed a new algorithm, described in the present work; this also differs from (and improves over) our previous technical report [7].

2 OUTLINE AND PROBLEM STATEMENT

In this section, we formalize our keyword search problem over a graph that we build by integrating data from various datasets, organized in different data models.

2.1 Integrated graph

We consider a set $\mathcal{M}$ of *data models*: relational (including SQL databases, CSV files etc.), RDF, JSON, HTML, XML, and text. A dataset D is an instance of one of these data models[3].

From a set $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ of datasets, we create an **integrated graph** $G = (N, E)$, where N is the set of nodes and E the set of edges. For instance, consider the dataset collection shown in Figure 1. Starting from the top left, in clockwise order, it shows: a table with assets of public officials, a JSON listing of France elected officials, an article from the newspaper Libération with entities highlighted, and a subset of the DBPedia RDF knowledge base.

Figure 2 shows the graph produced from the datasets in Figure 1. There are several observations to be made on this graph:

(*i*) The graph comprises four *dataset nodes* (the ones filled with yellow), one for each data source.

(*ii*) *All the internal structure present in the input datasets is preserved* in the graph: each RDF node became a node in the integrated graph, and each triple became an edge. A node is created for each map, array, and value in the JSON document. A node is created from each tuple, and from each attribute in the relational databases. Finally, a single node is created from the whole text document, which has no internal structure. When a text consists of more than one phrase, we segment it as a *sequence of phrases*, each of which is a node (child of the dataset node) to avoid overly large nodes that are hard to interpret by users.

(*iii*) *Entity nodes* (rounded-corners blue boxes) are extracted using Information Extraction (IE) techniques. Thus, in the example, nodes labeled "P. Balkany", "I. Balkany" are recognized as People, "Levallois-Perret" and "Centrafrique" are recognized as Locations, while "Areva" is an Organization. An extracted entity is added to the graph as a child of the node (leaf in an XML, HTML, JSON or text document; attribute value from a relational dataset; or RDF literal) from which it has been extracted.

(*iv*) *Equivalence edges* (solid red edges in Figure 1) connect nodes found in different datasets which are considered to refer to the same real-world entity. For instance, the three occurrences of "P. Balkany" are pairwise connected by edges with a *confidence* of 1.0. The confidence of the edges derived directly from the datasets, as explained above, is 1.0; we do not show it in the figure to avoid clutter. We say nodes connected by equivalence edges are *equivalent*.

(*v*) *Similarity edges* (dotted, curved red edge between "Central African Republic" and "Centrafrique" in Figure 1) connect nodes which are considered strongly similar but not equivalent. In our example, the two nodes have a similarity of 0.85, which is attached to the edge as confidence.

For efficiency, when k nodes are equivalent, we do not consider all the $\frac{k(k-1)}{2}$ edges; instead, one of the nodes (the first to be added to the graph - any other choice could be made) is designated the *representative* of all of them, and we store associated with each node, the ID of its representative.

The purpose of the equivalence and similarity edges is to **interconnect nodes within and across the datasets**; entity extraction prepares the ground for the same, since it creates nodes that may co-occur across data sources, e.g., entities mentioned in separate texts, such as "P. Balkany" in the figure. This increases the value and usefulness of the graph, since it allows to find connections which cannot be established based on any dataset taken separately. For instance, consider the question: *«What connections exist between "I. Balkany", "Africa", and "real estate"?»* This can be asked as a **three-keyword query** {"I. Balkany", "Africa", "Estate"}, for which an **answer** (a tree composed of graph edges) is shown as a light green highlight in Figure 1; the three nodes matching the respective keywords are shown in bold. This answer interconnects all four data sources.

[3]Our graph can also integrate other kinds of files, in particular PDF documents and spreadsheet files, by converting them to one or several instances of the above data models; as this is orthogonal wrt this paper, we delegate those details to [4].

Public officials transparency high authority (CSV)

Name	Owner	Location	Type
Dar Gyucy	P. Balkany	Marrakech	Real Estate
Moulin Cossy	I. Balkany	Giverny	Real Estate

dbpedia.org (RDF)

```
{
dbr:Marrakech
  dbr:name       "Marrakech"
  rdf:type       dbo:City ;
  dbo:country    dbr:Morrocco .
dbr:Morocco
  dbr:name       "Morocco"
  rdf:type       dbo:Country
  dbo:locatedIn dbr:Africa .
dbr:CentralAfricanRepublic
  dbr:name       "Central African Republic"
  dbo:locatedIn dbr:Africa .
}
```

National Directory of Elected Officials (JSON)

```
[{
   name: "Levallois-Perret",
   mayor: "P. Balkany",
   city-council: [
     {name: "I. Balkany"}, …    ]
}, …]
```

Libération – Nov. 13, 2014 (Text)

Balkany mineur de fonds

L'élu de Levallois-Perret est soupçonné d'avoir touché 5 millions de dollars de commission en 2009 grâce à son rôle d'intermédiaire entre Areva et la Centrafrique dans le dossier Uramin.
[…]

Figure 1: Sample dataset collection $\mathcal{D}$.

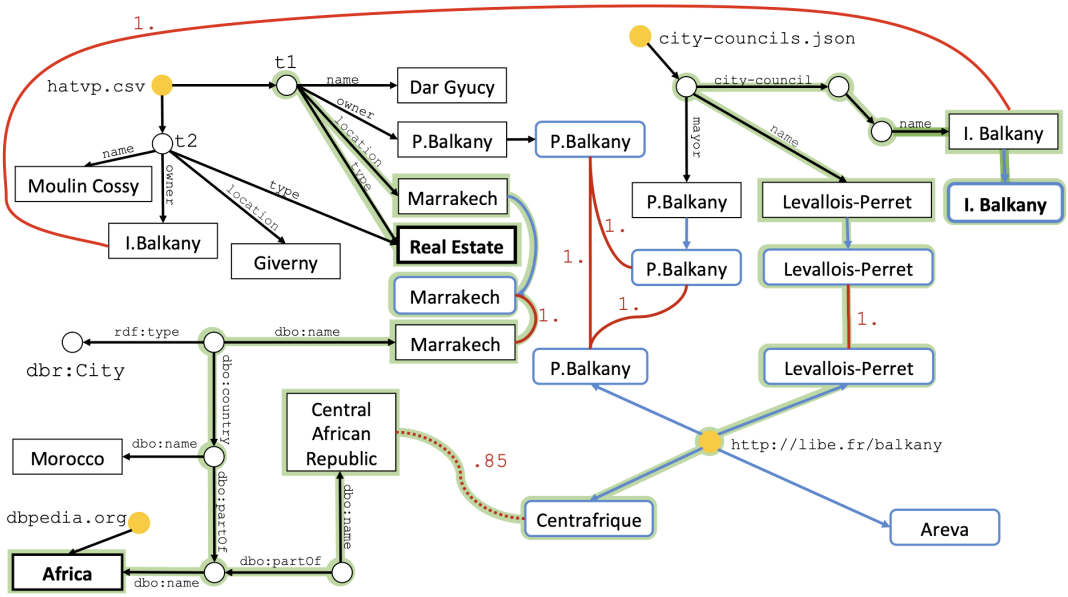


Figure 2: Integrated graph corresponding to the datasets of Figure 1.

We formalize this keyword search query problem below.

2.2 Search problem

Given our graph $G = (N, E)$, we denote by $\mathcal{L}$ the set of all the labels of G nodes, plus the special constant ϵ denoting the empty label. We denote by $\lambda(\cdot)$ a function assigning to each node and edge a label, which may be empty. As illustrated in Figure 2, internal nodes, which correspond, e.g., to a relational tuple, or to a JSON map or array, have an empty label.

Let W be the set of *keywords*, obtained by stemming the label set $\mathcal{L}$; a *search query* is a set of keywords $Q = \{w_1, ..., w_m\}$, where $w_i \in W$. We define an **answer tree** (AT, in short) as a set t of G edges which (i) together, form a tree (each node is reachable from any other through exactly one path), (ii) for each w_i, contain at least one node whose label matches w_i. Here, the edges are **considered undirected**, that is: $n_1 \xrightarrow{a} n_2 \xleftarrow{b} n_3 \xrightarrow{c} n_4$ is a sample AT, such that for all $w_i \in Q$, there is a node $n_i \in t$ such that $w_i \in \lambda(n_i)$.

We treat the edges of G as undirected when defining the AT in order to allow more query results, on a graph built out of heterogeneous content whose structure is not well-known to users. For instance, consider a query consisting of the keywords k_1, k_4 such that $k_1 \in \lambda(n_1)$ and $k_4 \in \lambda(n_4)$ on the four-nodes sample AT introduced above. If our ATs were restricted to the original direction

of G edges, the query would have no answer; ignoring the edge directions, it has one. One could easily extend the definition and the whole discussion in order to allow matches to also occur on edges (just enlarge $\mathcal{L}$ to also include the stemmed edge labels).

Further, we are interested in **minimal** answer trees, that is:

(1) Removing an edge from the tree should make it lack one or more of the query keywords w_i.
(2) If a query keyword w_i matches the label of more than one nodes in the answer tree, then all these matching nodes must be equivalent.

Condition (2) is specific to the graph we consider, originating in *several data sources connected by equivalence or similarity edges*. In classical graph keyword search problems, each query keyword is matched *exactly once* in an answer (otherwise, the tree is considered non-minimal). In contrast, our answer trees *may need to traverse equivalence edges*, and if w_i is matched by one node connected by such an edge, it is also matched by the other. For instance, consider the three-keyword query "Gyucy Balkany Levallois" in Figure 2: the keyword Balkany is matched by the two nodes labeled "P. Balkany" which are part of the answer.

As a counter-example to condition (2), consider the query "Balkany Centrafrique" in Figure 2, assuming the keyword Centrafrique is also matched in the label "Central African Republic"[4]. Consider the tree that connects a "P. Balkany" node with "Centrafrique", and also traverses the edge between "Centrafrique" and "Central African Republic": this tree is not minimal, thus it is not an answer. The intuition for rejecting it is that "Centrafrique" and "Central African Republic" may or may not be the same thing (we have a similarity, not an equivalence edge), therefore the query keyword "Centrafrique" is matched by two potentially different things in this answer, making it hard to interpret.

A direct consequence of minimality is that *in an answer, each and every leaf matches a query keyword.*

Several minimal answer trees may exist in G for a given query. We consider available a *scoring function* which assigns a higher value to more interesting answer trees (see Section 3). Thus, our problem can be stated as follows:

> Problem statement Given the graph G built out of the datasets $\mathcal{D}$ and a query Q, return the k highest-score minimal answer trees.

An AT may potentially span over the whole graph, (also) because it can traverse G edges in any direction; this makes the problem challenging.

Discussion: degraded answers. In some cases, a query may have no answer (as defined above) on a given graph, yet if one is willing to drop the second condition concerning nodes matching the same query keyword, an answer tree could be found. For instance, consider a graph of the form $a_1 \xrightarrow{l} b_1 \xrightarrow{m} b_2 \xrightarrow{n} c_1$, such that b_1 is *not equivalent to* b_2, and the query $\{a, b, c\}$, such that the keyword a matches the node a_1, b matches b_1 and b_2 and c matches c_1. Given our definition of answers above, this query has no answer, because b matches the two nodes b_1 and b_2.

[4]This may be the case using a more advanced indexing system that includes some natural language understanding, term dictionaries etc.

If we removed condition (2), we could accept such an answer, which we call *degraded*, since it is harder to interpret for users (lacking one clearly identified node for each keyword). One could then generalize our problem statement into: (*i*) solve the problem stated above, and (*ii*) only if there are no answers, find the top-k degraded answers (if they exist). We do not pursue degraded answer search further in this paper, and focus instead on finding those defined above.

2.3 Search space and complexity

The problem that we study is related to the (Group) Steiner Tree Problem, which we recall below.

Given a graph G with weights (costs) on edges, and a set of m nodes $n_1, \ldots, n_m$, the *Steiner Tree Problem (STP)* [14] consists of finding the smallest-cost tree in G that connects all the nodes together. We could answer our queries by solving one STP problem for each combination of nodes matching the keywords $w_1, \ldots, w_m$. However, there are several obstacles left: ($\diamond$) STP is a known NP-hard problem in the size of G, denoted $|G|$; ($\triangleright$) as we consider that each edge can be taken in the direct or reverse direction, this amounts to "doubling" every edge in G. Thus, our search space is $2^{|G|}$ **larger than the one of the STP, or that considered in similar works**, discussed in Section 6. This is daunting even for small graphs of a few hundred edges; ($\triangleleft$) we need the k smallest-cost trees, not just one; ($\circ$) each keyword may match several nodes, not just one.

The closely related *Group STP* (GSTP, in short) [14] is: given m *sets of nodes* from G, find the minimum-cost subtree connecting one node from each of these subtrees. GSTP does not raise the problem ($\circ$), but still has all the others.

In conclusion, the complexity of the problem we consider is extremely high. Therefore, solving it fully is unfeasible for large and/or high-connectivity graphs. Instead, our approach is:

- *Attempt to find all answers from the smallest* (fewest edges) *to the largest.* Enumerating small trees first is both a practical decision (we use them to build larger ones) and fits the intuition that we shouldn't miss small answers that a human could have found manually. However, as we will explain, we still "opportunistically" build some trees before exhausting the enumeration of smaller ones, whenever this is likely to lead faster to answers. The strategy for choosing to move towards bigger instead of smaller tress leaves rooms for optimizations on the search order.
- *Stop at a given time-out or when m answers have been found*, for some $m \geq k$;
- *Return the k top-scoring answers* found.

3 SCORING ANSWER TREES

We now discuss how to evaluate the quality of an answer. Section 3.1 introduces the general notion of score on which we base our approach. Section 3.2 describes one particular metric we attach to edges in order to instantiate this score, finally Section 3.3 details the actual score function we used.

3.1 Generic score function

We have configured our problem setting to allow *any scoring function*, which enables the use of different scoring schemes fitting the

requirements of different users. As a consequence, this approach allows us to study the interaction of the scoring function with different properties of the graph. For instance, we are currently investigating the possibility to *learn* what makes an answer interesting for a user, so that we may return customized answers to each user.

Given an answer tree t to a query Q, we consider a score function consisting of (at least) the following two components:

- The *matching score* $ms(t)$, which reflects the quality of the answer tree, that is, how well its leaves match the query terms.
- The *connection score* $cs(t)$, which reflects the quality of the tree connecting the edges. Any formula can be used here, considering the number of edges, the confidence or any other property attached to edges, or a query-independent property of the nodes, such as their PageRank or betweenness centrality score etc.

The score of t for Q, denoted $s(t)$, is computed as a combination of the two independent components $ms(t)$ and $cs(t)$. Popular combinations functions (a weighted sums, or product etc.) are monotonous in both components, however, our framework does not require it. Finally, both $ms(t)$ and $cs(t)$ can be tuned based on a given user's preferences, to personalize the score, or make them evolve in time through user feedback etc.

3.2 Edge specificity

We now describe a metric on edges, which we used (through the connection score $cs(t)$) to favor edges that are "rare" for both nodes they connect. This metric was inspired by our experiments with real-world data sources, and it helped return interesting answer trees in our experience.

For a given node n and label l, let $N^l_{\rightarrow n}$ be the number of l-labeled edges entering n, and $N^l_{n\rightarrow}$ the number of l-labeled edges exiting n. The **specificity** of an edge $e = n_1 \xrightarrow{l} n_2$ is defined as:

$$s(e) = 2/(N^l_{n_1\rightarrow} + N^l_{\rightarrow n_2}).$$

$s(e)$ is 1.0 for edges that are "unique" for both their source and their target, and decreases when the edge does not "stand out" among the edges of these two nodes. For instance, the city council of Levallois-Perret comprises only one mayor (and one individual cannot be mayor of two cities in France, because he has to inhabit the city where he runs for office). Thus, the edge from the city council to P. Balkany has a specificity of $2/(1.0 + 1.0) = 1.0$. In contrast, there are 54 countries in Africa (we show only two), and each country is in exactly one continent; thus, the specificity of the dbo:partOf edges in the DBPedia fragment, going from the node named Morocco (or the one named Central African Republic) to the node named Africa is $2/(1 + 54) \simeq .036$.

Specificity computation. When registering the first dataset D_1, computing the specificity of its edges is trivial. However, when registering subsequent datasets D_2, D_3 etc., if some node, say $n_2 \in D_2$ is found to be equivalent to a node $n_1 \in D_1$, *all* the D_1 edges adjacent to n_1 *and* the D_2 edges adjacent to n_2 should be reflected in the specificity of *each* of these edges. Thus, in particular, the specificity of D_1 edges needs to be *recomputed* when a node in a source added after D_1 is equivalent to one of its nodes.

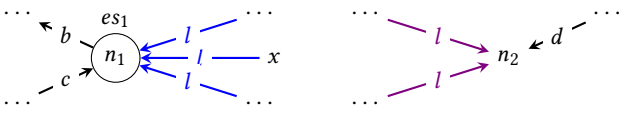


Figure 3: Illustration for specificity (re)computation. The specificity of the edge $x \xrightarrow{l} n_1$, $s(e)$ is initially computed out of the blue edges; when n_2 joins the equivalence set es_1, it is recomputed to also reflect the violet edges.

A *naïve approach* would be: when the edges of D_2 are traversed (when we add this dataset to the graph), re-traverse the edges of n_1 in D_1 in order to (re)compute their specificity. However, that would be quite inefficient.

Instead, below, we describe an *efficient incremental algorithm* to compute specificity. We introduce two notations. For any edge e, we denote $N^e_{\rightarrow\bullet}$, respectively $N^e_{\circ\rightarrow}$, the two numbers out of which the specificity of e has been *most recently* computed[5]. Specifically, $N^e_{\rightarrow\bullet}$ counts l-labeled edges incoming to the target of e, while $N^e_{\circ\rightarrow}$ counts l-labeled edges outgoing the source of e. In Figure 3, if e is the edge $x \xrightarrow{l} n_1$, then $N^e_{\rightarrow\bullet} = 3$ (blue edges) and $N^e_{\circ\rightarrow} = 1$, thus $s(e) = 2/4 = .5$.

Let $n_1 \in D_1$ be a node, es_1 be the set of all nodes equivalent to n_1, and $n_2 \in D_2$ be a node in a dataset we currently register, and which has just been found to be equivalent to n_1, also.

Further, let l be a label of an edge incoming or outgoing (any) node from es_1, and/or n_2. We denote by $N^l_{\rightarrow es_1}$ the sum $\sum_{n\in es_1}(N^l_{\rightarrow n})$ and similarly by $N^l_{es_1\rightarrow}$ the sum $\sum_{n\in es_1}(N^l_{n\rightarrow})$; they are the numbers of l-labeled outgoing (resp., incoming) l-labeled edges of any node in es_1. When n_2 joins the equivalence set es_1 of n_1 (see Figure 3):

(1) If $N^l_{\rightarrow es_1} \neq 0$ and $N^l_{\rightarrow n_2} \neq 0$, the specificity of every l-labeled edge e *incoming* either a node in es_1 or the node n_2 must be recomputed.
Let e be such an *incoming* edge labeled l. When n_2 is added to the set es_1, the specificity of e becomes $2/((N^e_{\rightarrow\bullet} + N^l_{\rightarrow n_2}) + N^e_{\circ\rightarrow})$, to reflect that n_2 brings more incoming l-labeled edges. This amounts to $2/(3 + 2 + 1) = .33$ in Figure 3: the violet edges have joined the blue ones. Following this adjustment, the numbers out of which e's specificity has been most recently computed are modified as follows: $N^e_{\rightarrow\bullet}$ becomes $N^e_{\rightarrow\bullet} + N^l_{\rightarrow n_2}$, thus $3 + 2 = 5$ in Figure 3; $N^e_{\circ\rightarrow}$ remains unchanged.
(2) If $N^l_{\rightarrow es_1} = 0$ and $N^l_{\rightarrow n_2} \neq 0$, the specificity of every l-labeled edge e *incoming* n_2 does not change when n_2 joins the equivalence set es_1.
(3) If $N^l_{\rightarrow es_1} \neq 0$ and $N^l_{\rightarrow n_2} = 0$, the newly added node n_2 does not change the edges adjacent to the nodes of es_1, nor their specificity values.

The last two cases, when $N^l_{es_1\rightarrow} \neq 0$ and $N^l_{n_2\rightarrow} \neq 0$, respectively, $N^l_{es_1\rightarrow} = 0$ and $N^l_{n_2\rightarrow} \neq 0$, are handled in a similar manner.

The above method only needs, for a given node n_2 newly added to the graph, and label l, the number of edges adjacent to n_2 in its dataset, and the *number* of l edges adjacent to a node equivalent

[5]This can be either during the first specificity computation of e, or during a recomputation, as discussed below.

to n_2. Unlike the naïve specificity computation method, it does not need to actually *traverse* these edges previously registered edges, making it more efficient.

Concretely, for each edge $e \in E$, we store three attributes: $N^e_{\rightarrow\bullet}$, $N^e_{\circ\rightarrow}$ and s, the last-computed specificity, and we update $N^e_{\rightarrow\bullet}$, $N^e_{\circ\rightarrow}$ as explained above.

3.3 Concrete score function

In our experiments, we used the following score function.

For an answer t to the query Q, we compute the matching score $ms(t)$ as the *average*, over all query keywords w_i, of the similarity between the t node matching w_i and the keyword w_i itself; we used the edit distance.

We compute the connection score $cs(t)$ based on edge confidence, on one hand, and edge specificity on the other. We *multiply* the confidence values, since we consider that uncertainty (confidence < 1) multiplies; and we also *multiply* the specificities of all edges in t, to discourage many low-specificity edges. Specifically, our score is computed as:

$$score(t, Q) = \alpha \cdot ms(t, Q) + \beta \cdot \prod_{e \in E} c(e) + (1 - \alpha - \beta) \cdot \prod_{e \in E} s(e)$$

where α, β are parameters of the system such that $0 \leq \alpha, \beta < 1$ and $\alpha + \beta \leq 1$.

3.4 Orthogonality between the score and the algorithm

Before we describe the search algorithm, we make a few more remarks on the connection between the score function and the search algorithm.

We start by considering the classical Steiner Tree and Group Steiner Tree Problems (Section 2.3). These assume that the tree cost is **monotonous**, that is: for any query Q and all trees $T, \hat{T}$ where T is a subtree of $\hat{T}$ it follows that the cost of T is higher (in our terminology, its score is lower) than the cost of $\hat{T}$. This is naturally satisfied if the cost is the addition of edge weights. In contrast, the score, in its general form (Section 3.1), and in particular our concrete one (Section 3.3), is **not monotonous**, as illustrated in Figure 4, where on each edge, c is the confidence and s is the specificity. Denoting T the four-edge tree rooted in n_1, the connection score $cs(T) = \beta + (1-\alpha-\beta)\cdot(.5)^4$, while $cs(T') = \beta \cdot .5 + (1-\alpha-\beta)(.5)^4 \cdot .25$. If we assume $\alpha = \beta = \frac{1}{3}$, then $cs(T) = \frac{1}{3}(1 + (.5)^4) \simeq .35$ while $cs(T') = \frac{1}{3} \cdot (.5 + (.5)^5) \simeq .17$, which is clearly smaller. Assuming T' has the same matching score as T, the global score of T' is smaller than that of T, contradicting the monotonicity assumption.

Another property sometimes assumed by score functions is the called **optimal substructure**, that is: the best solution for a problem of size p is part of the best solution for a problem of size $p + 1$ that is an extension of p, for some problem size p. When this holds, the problem can be efficiently solved in a dynamic programming fashion. However, STP does not enjoy this property: the smallest-cost tree connecting two nodes n_1, n_2 is not necessarily part of the smallest-cost tree that connects n_1, n_2, n_3 (and the same holds for GSTP). Some existing algorithms also assume a variant of the optimal substructure property (see Section 6). In contrast, our score function (both in its general and its concrete form) does not ensure such favorable properties. This is why the search algorithm we describe next has to find as many answers as possible, as quickly as possible.

4 ANSWERING KEYWORD QUERIES

We now present our approach for computing query answers, based on the integrated graph.

4.1 Grow and Merge

Our first algorithm uses some concepts from the prior literature [10, 18] while exploring many more trees. Specifically, it starts from the sets of nodes $N_1, \ldots, N_m$ where the nodes in N_i all match the query keyword w_i; each node $n_{i,j} \in N_i$ forms a one-node partial tree. For instance, in Figure 2, one-node trees are built from the nodes with boldface text, labeled "Africa", "Real Estate" and "I. Balkany". Two transformations can be applied to form increasingly larger trees, working toward query answers:

- Grow(t, e), where t is a tree, e is an edge *adjacent to the root* of t, and e does not close a loop with a node in t, creates a new tree t' having all the edges of t plus e; the root of the new tree is the other end of the edge e. For instance, starting from the node labeled "Africa", a Grow can add the edge labeled `dbo:name`.
- Merge(t_1, t_2), where t_1, t_2 are trees with the same root, whose other nodes are disjoint, and matching disjoint sets of keywords, creates a tree t'' with the same root and with all edges from t_1 and t_2. Intuitively, Grow moves away from the keywords, to explore the graph; Merge fuses two trees into one that matches more keywords than both t_1 and t_2.

In a *single-dataset* context, Grow and Merge have the following properties. (gm_1) Grow alone is **complete** (guaranteed to find all answers) for $k = 1, 2$ only; for higher k, Grow and Merge together are complete. (gm_2) Using Merge steps helps find answers faster than using just Grow [18]: partial trees, each starting from a leaf that matches a keyword, are merged into an answer as soon as they have reached the same root. (gm_3) An answer can be found through **multiple combinations of Grow and Merge**. For instance, consider a linear graph $n_1 \rightarrow n_2 \rightarrow \ldots n_p$ and the two-keyword query $\{a_1, a_p\}$ where a_i matches the label of n_i. The answer is obviously the full graph. It can be found: starting from n_1 and applying $p - 1$ Grow steps; starting from n_p and applying $p-1$ Grow steps; and in $p - 2$ ways of the form Merge(Grow(Grow…), Grow(Grow…)), each merging in an intermediary node $n_2, \ldots, n_{p-1}$. These are all the same according to our definition of an answer (Section 2.2), which does not distinguish a root in an answer tree; this follows users' need to know how things are connected, and for which the tree root is irrelevant.

4.2 Adapting to multi-datasets graphs

The changes we brought for our harder problem (bidirectional edges and multiple interconnected datasets) are as follows.

1. Bidirectional growth. We allow Grow to traverse an edge both going from the source to the target, and going from the target to the source. For instance, the type edge from "Real Estate" to <tuple1> is

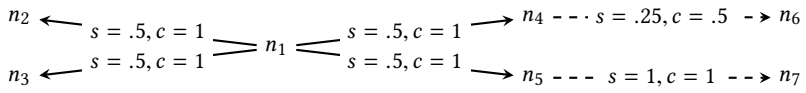


Figure 4: Example (non-monotonicity of the tree score). T is the four-edges tree rooted in n_1.

traversed target-to-source, whereas the location edge from <tuple1> to "Real Estate" is traversed source-to-target.

2. Many-dataset answers. As defined in a single-dataset scenario, GROW and MERGE do not allow to connect multiple datasets. To make that possible, we need to enable one, another, or both to also traverse similarity and equivalence edges (shown in solid or dotted red lines in Figure 2. We decide to simply extend GROW to allow it to traverse not just data edges, but also *similarity* edges between nodes of the same or different datasets. We handle *equivalence* edges as follows:

2.a Naïve solution: GROW-to-equivalent. The simplest idea is to allow GROW to also add an equivalence edge to the root of a tree. However, this can be very inefficient. Consider three equivalent nodes m, m' and m'', e.g., the three "P. Balkany" nodes in Figure 2: a GROW step could add one equivalence edge, the next GROW could add another on top of it etc. More generally, for a group of p equivalent nodes, from a tree rooted in one of these nodes, 2^p trees would be created just by GROW. In our French journalistic datasets, some entities, e.g. "France", are very frequent, leading to high p; exploring 2^p subtrees every time we reach a "France" node is extremely expensive[6].

2.b GROW-to-representative To avoid this, we devise a third algorithmic step, called *GROW-to-representative* (GROW2REP), as follows. Let t be a partial tree developed during the search, rooted in a node n, such that the representative of n (recall Section 2.1) is a node $n_{rep} \neq n$. GROW2REP creates a new tree by adding to t the edge $n \xrightarrow{\equiv} n_{rep}$; this new tree is rooted in n_{rep}. If n is part of a group of p equivalent nodes, only *one* GROW2REP step is possible from t, to the unique representative of n; GROW2REP does not apply again on GROW2REP(t), because the root of this tree is n_{rep}, which is its own representative.

Together, GROW, GROW2REP and MERGE enable finding answers that span multiple data sources, as follows:

- GROW allows exploring data edges within a dataset, and similarity edges within or across datasets;
- GROW2REP goes from a node to its representative when they differ; the representative may be in a different dataset;
- MERGE merges trees with a same root: when that root is the representative of a group of p equivalent nodes, this allows connecting partial trees, including GROW2REP results, containing nodes from different datasets. Thus, MERGE can build trees spanning multiple datasets.

[6]Note that *similarity* edges do not raise the same problem, because in our graph we only have such edges if the similarity between two nodes is above a certain threshold τ. Thus, if a node n_1 is at least τ-similar to n_2, and n_2 is at least τ-similar to n_3, n_1 may be at least τ-similar to n_3, or not. This leads to much smaller groups of similar nodes, than the groups of equivalent nodes we encountered.

One potential performance problem remains. Consider again p equivalent nodes $n_1, \ldots, n_p$; assume without loss of generality that their representative is n_1. Assume that during the search, a tree t_i is created rooted in each of these p nodes. GROW2REP applies to all but the first of these trees, creating the trees $t'_2, t'_3, \ldots, t'_p$, all rooted in n_1. Now, MERGE can merge any pair of them, and can then repeatedly apply to merge three, then four such trees etc., as they all have the same root n_1. The exponential explosion of GROW trees, avoided by introducing GROW2REP, is still present due to MERGE!

We solve this problem as follows. Observe that in an answer, a *path of two or more equivalence edges* of the form $n_1 \xrightarrow{\equiv} n_2 \xrightarrow{\equiv} n_3$ such that *a node internal to the path*, e.g. n_2, *has no other adjacent edge*, even if allowed by our definition, is *redundant*. Intuitively, such a node brings nothing to the answer, since its neighbors, e.g., n_1 and n_3, could have been connected directly by a single equivalence edge, thanks to the transitivity of equivalence. We call *non-redundant* an answer that does not feature any such path, and decide to **search for non-redundant answers** only.

The following properties hold on non-redundant answers:

PROPERTY 1. *There exists a graph G and a k-keyword query Q such that a non-redundant answer contains $k-1$ adjacent equivalence edges (edges that, together, form a single connected subtree).*

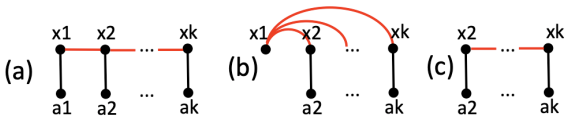


Figure 5: Sample answer trees for algorithm discussion.

We prove this by exhibiting such an instance. Let G be a graph of $2k$ nodes shown in Figure 5 (a), such that all the x_i are equivalent, and consider the k-keyword query $Q = \{a_1, \ldots, a_k\}$ (each keyword matches exactly the respective a_i node). An answer needs to traverse all the k edges from a_i to x_i, and then connect the nodes $x_i, \ldots, x_k$; we need $k-1$ equivalence edges for this.

Next, we show:

PROPERTY 2. *Let t be a non-redundant answer to a query Q of k keywords. A group of adjacent equivalence edges contained in t has at most $k-1$ edges.*

We prove this by induction over k. For $k = 1$, each answer has 1 node and 0 edge (trivial case).

Now, consider this true for k and let us prove it for $k+1$. Assume by contradiction that a non-redundant answer t_Q to a query Q of $k+1$ keywords comprises $k+1$ adjacent equivalence edges. Let Q' be the query having only the first k keywords of Q, and t' be a subtree of t that is a non-redundant answer to Q':

- t' exists, because t connects all Q keywords, thus also the Q' keywords;
- t' is non-redundant, because its edges are also in the (non-redundant) t.

By the induction hypothesis, t' has at most $k-1$ adjacent equivalence edges. This means that there are *two adjacent equivalent edges* in $t \setminus t'$.

(1) If these edges, together, lead to two distinct leaves of t, then t has *two* leaves not in t'. This is not possible, because by definition of an answer, t has $k+1$ leaves (each matching a keyword) and similarly t' has k leaves.
(2) It follows, then, that the two edges lead to a single leaf of t, therefore the edges form a redundant path. This contradicts the non-redundancy of t, and concludes our proof.

Property 2 gives us an important way to control the exponential development of trees due to p equivalent nodes. Grow, Grow2Rep and Merge, together, can generate trees with up to k (instead of $k-1$) adjacent equivalence edges. This happens because Grow2Rep may "force" the search to visit the representative of a set of k equivalent nodes (see Figure 5(b), assuming x_1 is the representative of all the equivalent x_is, and the query $\{a_2, \ldots, a_k\}$). The resulting answer may be redundant, if the representative has no other adjacent edges in the answer other than equivalence edges. In such cases, in a **post-processing step**, we remove from the answer the representative and its equivalence edges, then reconnect the respective equivalent nodes using $k-1$ equivalence edges. This guarantees obtaining a non-redundant tree, such as the one in Figure 5(c).

4.3 The GAM algorithm

We now have the basic exploration steps we need: Grow, Grow2Rep and Merge. In this section, we explain how we use them in our integrated keyword search algorithm.

We decide to apply in sequence: one Grow or Grow2Rep (see below), leading to a new tree t, immediately followed by all the Merge operations possible on t. Thus, we call our algorithm **Grow and Aggressive Merge** (GAM, in short). We merge aggressively in order to detect as quickly as possible when some of our trees, merged at the root, form an answer.

Given that every node of a currently explored answer tree can be connected with several edges, we need to decide which Grow (or Grow2Rep) to apply at a certain point. For that, we use a **priority queue** U in which we add (tree, edge) entries: for Grow, with the notation above, we add the (t, e) pair, while for Grow2Rep, we add t together with the equivalence edge leading to the representative of t's root. In both cases, when a (t, e) pair is extracted from U, we just extend t with the edge e (adjacent to its root), leading to a new tree t_G, whose root is the other end of the edge e. Then we aggressively merge t_G with all compatible trees explored so far, finally we read from the graph the (data, similarity or equivalence) edges adjacent to t_G's root and add to U more (tree, edge) pairs to be considered further during the search. The algorithm then picks the highest-priority pair in U and reiterates; it stops when U is empty, at a timeout, or when a maximum number of answers are found (whichever comes first).

The last parameter impacting the exploration order is the priority used in U: at any point, U gives the highest-priority (t, e) pair, which determines the operations performed next.

(1) Trees matching *many query keywords* are preferable, to go toward complete query answers;
(2) At the same number of matched keywords, *smaller trees* are preferable in order not to miss small answers;
(3) Finally, among (t_1, e_1), (t_2, e_2) with the same number of nodes and matched keywords, we prefer the pair with the *higher specificity edge*.

Algorithm details Beyond the priority queue U described above, the algorithm also uses a *memory of all the trees explored*, called E. It also organizes all the (non-answer) trees into a map K in which they can be accessed by the subset of query keywords that they match. The algorithm is shown in pseudocode in Figure 6, following the notations introduced in the above discussion.

While not shown in Figure 6 to avoid clutter, the algorithm *only develops minimal trees* (thus, it only finds minimal answers). This is guaranteed:

- When creating Grow and Grow2Rep opportunities (steps 3 and 4d): we check not only that the newly added does not close a cycle, but also that the matches present in the new tree satisfy our minimality condition (Section 2.2).
- Similarly, when selecting potential Mergecandidates (step 4(c)iiiA).

5 EXPERIMENTAL EVALUATION

We implemented our approach in the **ConnectionLens** prototype, available online at https://gitlab.inria.fr/cedar/connectionlens, which we used to experimentally evaluate the performance of our algorithms. This section presents the results that we obtained by using synthetic graphs, which are similar to the real-world datasets that we have obtained. First, we describe the hardware and software setup that we used to run our experiments, and then we give our findings for various combinations of amount of keywords and graph sizes.

5.1 Hardware and software setup

We conducted our experiments on a server equipped with 2x10-core Intel Xeon E5-2640 CPUs clocked at 2.40GHz, and 128GB DRAM. The graph is constructed following the approach described in [4] and we used Postgres 9.6.5 to store and query the graph for nodes, edges and labels. The search algorithms are implemented in a Java application which communicates with the database over JDBC, whereas it also maintains an in-memory cache. Every time that the search algorithm needs information about a node, it first looks into the cache, and if the requested information is not there, it is directly retrieved from the database and then stored in the cache. To avoid any effects of the cache replacement algorithm, in our experiments we set the cache to be large enough to include all the information that has been retrieved from the database.

Synthetic datasets For controlled experiments, we generated different types of (RDF) graphs. The first type is a *line graph*, which the simplest model that we can use. In the line graph, every node is connected with two others, having one edge for each node, except two nodes which are connected with only one. By using the line graph,

Procedure **process**(tree t)

- if t is not already in E
- then
  - add t to E
  - if t has matches for all the query keywords
  - then post-process t if needed; output the result as an answer
- else insert t into K

Algorithm **GAMSearch**(query $Q = \{w_1, w_2, \ldots, w_k\}$)

(1) For each w_i, $1 \leq i \leq k$
   - For each node n_i^j matching w_i, let t_i^j be the 1-node tree consisting of n_i^j; process(t_i^j)

(2) Initial MERGE*: try to merge every pair of trees from E, and process any resulting answer tree.

(3) Initialize U (empty so far):
   (a) Create GROW opportunities: Insert into U the pair (t, e), for each $t \in E$ and e a data or similarity edge adjacent to t's root.
   (b) Create GROW2REP opportunities: Insert into U the pair $(t, n \rightarrow n_{rep})$ for each $t \in E$ whose root is n, such that the representative of n is $n_{rep} \neq n$.

(4) While (U is not empty)
   (a) Pop out of U the highest-priority pair (t, e).
   (b) Apply the corresponding GROW or GROW2REP, resulting in a new tree t''; process(t'').
   (c) If t'' was not already in E, agressively MERGE:
      (i) Let NT be a set of new trees obtained from the MERGE (initially $\emptyset$).
      (ii) Let $\mathbf{p_1}$ be the keyword set of t''
      (iii) For each keyword subset $\mathbf{p_2}$ that is a key within K, and such that $\mathbf{p_1} \cap \mathbf{p_2} = \emptyset$
         (A) For each tree t^i that corresponds to $\mathbf{p_2}$, try to merge t'' with t^i. Process any possible result; if it is new (not in E previously), add it to NT.
   (d) Re-plenish U (add more entries in it). This is performed as in step 3 but based on NT (not on E).

Figure 6: Outline of GAM algorithm

we clearly show the performance of GROW and MERGE operations with respect to the size of the graph. The second type is a *chain graph*, which is the same as the line graph, but instead of one edge connecting every pair of nodes, we have two. We use this type to show the performance of the algorithm as we double the amount of edges of the line graph and we give more options to the GROW and the MERGE algorithms. The third type is the *star graph*, where we have several line graphs connected through a strongly connected cluster of nodes with a representative. We use this type to show the performance of GROW2REP, by placing the query keywords on different line graphs. The fourth type is a random graph based on the *Barabasi-Albert* (BA, in short) model [1], which generates scale-free networks with only a few nodes (referred to as hubs) of the graph having much higher degree than the rest. The graph in this model is created in a two-staged process. During the first stage, a network of some nodes is created. Then, during the second stage, new nodes are inserted in the graph and they are connected to nodes created during the first stage. At the second stage, we can control how many connections every node will have to the ones created at the first stage. By setting that every node created at the second stage to be connected with exactly one node created at the first stage, we have observed that we can construct graphs which are similar to the real-world ones, and therefore we tune our model accordingly.

Real-world dataset The real-world dataset that we used is based on data that we have obtained from journalists with whom we collaborate. Our dataset combines information on French politics, which we obtained by crawling the web pages of a French newspaper, as we explain in the corresponding Section.

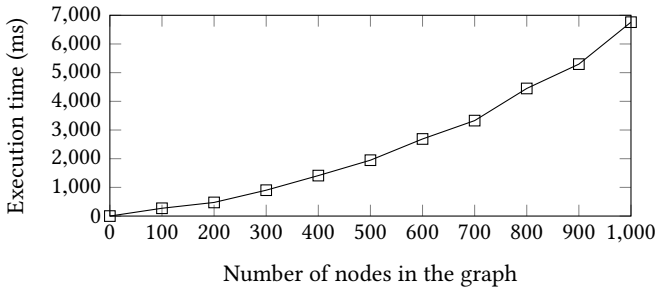


Figure 7: Line graph execution time

For the microbenchmarks, we report the time needed for our system to return the first answer, as well as the time for all answers. For the macrobenchmarks, we only report the time to return all the answers, as we do not have full control over the graphs and, hence, it is hard to draw meaningful conclusions and explain them relying on the whole graphs. Finally, we set an upper bound to the overall execution time at 120 seconds, which is applied to all the experiments that we performed.

5.2 Querying synthetic datasets

Figure 7 shows the execution time of our algorithm when executing a query with two keywords on a line graph, as we vary the number of nodes of the graph. We place the keywords on the two "ends" of the graph to show the impact of the distance on the execution time. The performance of our algorithm is naturally affected by the size of the graph, as it generates $2 * N$ answer trees, where N is the number of nodes. Given that this is a line graph, there is only one answer, which is the whole graph, and, therefore, the time to find the first answer is also the overall execution time.

Figure 8 shows the execution of our algorithm on a chain graph. Specifically, Figure 8a shows the time elapsed until the first answer is found, whereas Figure 8b shows the overall execution time. The execution times reported in Figure 8a are almost the same, as the size of the graph increases slowly. On the other hand, the overall execution times increase at a much higher (exponential) rate, as shown in Figure 8b, where the y axis has a logarithmic scale. The reason is that every pair of nodes is connected with two edges, which increases the amount of answers exponentially with the amount of nodes in the graph.

Similar to the chain graph, in Figure 9 we report the execution time until our algorithm finds the first and all answers (left and right hand side, respectively). Given that we use keywords which are placed in two different lines connected through the center of the graph, the algorithm has to use Grow2Rep, whereas in the previous cases it only had to use Grow and Merge. The number of branches, depicted on the x axis of Figure 9, corresponds to the number of line graphs connected in the star. Each line graph has 10 nodes and we place the query keywords at the extremities of two different line graphs. Given that our algorithm will have to check all possible answers, it follows that the number of merges is exponential to the number of branches, that is $O(2^K)$, where K is the number of branches. This behaviour is clearly shown in both parts of Figure 9, where on the y axis (in logarithmic scale) we show the times to find the first, and, respectively, all answers. Above 12 branches, the timeout of 120 seconds that we have set is hit and, thus, search is terminated, as shown in Figure 9b.

Figure 10 depicts the performance of our algorithm when considering the Barabasi-Albert graph model. In this experiment, we keep the graph with 2000 nodes fixed and we vary the position of two keywords, by choosing nodes which have a distance, as given in the x axis; note the logarithmic y axis. Due to the fact that the graph is randomly generated within the BA model, we note some irregularity in the time to the first solution, which however grows at a moderate pace as the distance between the keyword node grows. The overall relation between the time to the first solution and the total time confirms that the search space is very large but that most of the exploration is not needed, since the first solution is found quite fast.

5.3 Querying a real-world dataset

This Section includes the results that we obtained by running our algorithm on real-world data. Our dataset is a corpus of 462 HTML articles (about 6MB) crawled from the French online newspaper Mediapart with the search keywords "gilets jaunes" (yellow vests, a protest movement in France over the last year). We built a graph using these articles which consists of 90626 edges and 65868 nodes, out of which 1525 correpond to people, 1240 to locations and 1050 to organizations. We query the graph using queries of one, two and three different keywords.

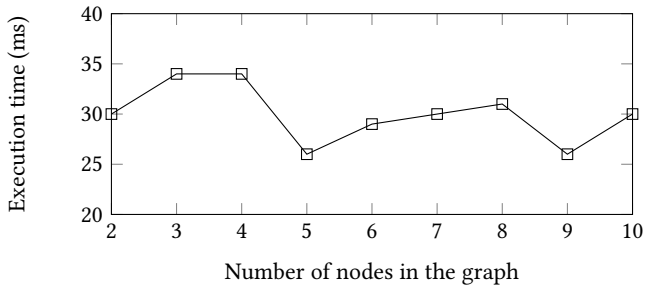
We report our findings in Table 1. The results are not given as a basis comparison, rather than as a proof of concept. Nevertheless, there are several interesting observations to be made. First, the amount of answers for every query is generally larger than 1. We allow several results, as the end users (in our case, the investigative journalists) need to see different connections to reach to potentially interesting conclusions. Second, there are queries where several answers are found, and the execution is interrupted due to the threshold. We allow the user to set the threshold, based on the results returned every time. Third, the answers returned to the user are significantly less than the answer trees discovered, showing the impact of minimality as a requirement for returning an answer.

6 RELATED WORK AND CONCLUSIONS

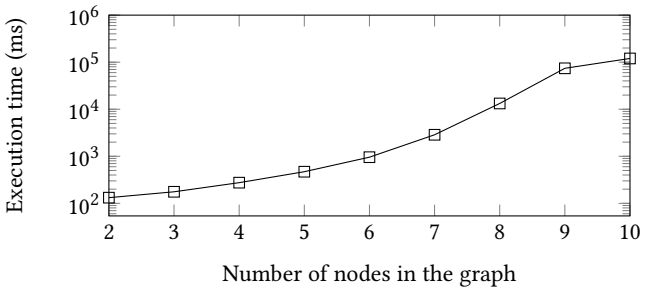
Keyword search (KS, in short) is the method of choice for searching in unstructured (typically text) data, and it is also the best search method for novice users, as witnessed by the enormous success of keyword-based search engines. As databases grew larger and more complex, KS has been proposed as a method for searching *also* in structured data [31], when users are not perfectly familiar with the data, or to get answers enabled by different tuple connection networks. For **relational** data, in [19] and subsequent works, tuples are represented as nodes, and two tuples are interconnected only through primary key-foreign key pairs. The graphs that result are thus quite uniform, e.g., they consist of "Company nodes", "Employee nodes" etc. The same model was considered in [8, 27, 29, 30, 32]; [27] also establishes links based on similarity (or equality) of constants appearing in different relational attributes. As explained in Section 2.3, our problem is (much) harder since our trees can traverse edges in both directions, and paths can be (much) longer than those based on PK-FK alone. [30] proposes to incorporate user feedback through active learning to improve the quality of answers in a relational data integration setting. We are working to devise such a learning-to-rank approach for our graphs, also.

KS has also been studied in **XML documents** [17, 25]. Here, an answer is defined as a subtree of the original document, whose leaves match the query keywords. This problem is much easier than the one we face, since: (*i*) an XML document is a tree, guaranteeing just one connection between any two nodes; in contrast, there can be any number of such connections in our graphs; (*ii*) the maximum size of an answer to a k-keywords query is $k \cdot h$ where h, the height of an XML tree, is almost always quite small, e.g., 20 is considered "quite high"; in contrast, with our bi-directional search, the bound is $k \cdot D$ where D is the diameter of our graph - which can be enormously larger.

Our Grow and Merge steps are borrowed from [10, 18], which address KS for **graphs**, assuming optimal-substructure which does not hold for us, and single-direction edge traversal. For **RDF graphs** [13, 22] traverse edges in their direction only; moreover, [22] also make strong assumptions on the graph, e.g., that all non-leaf nodes have types, and that there are a small number of types (regular graph).

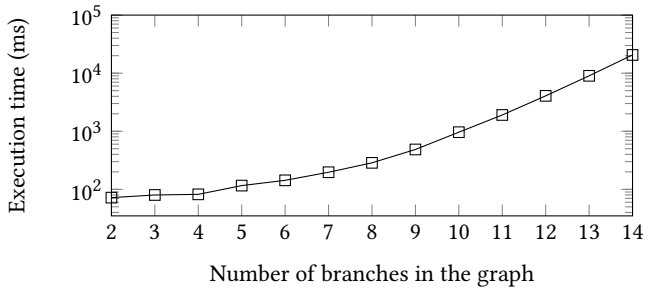


(a) Time to find the first answer

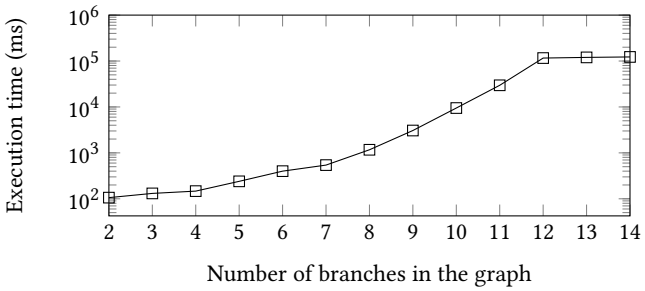


(b) Time to find all answers

Figure 8: Chain graph execution time



(a) Time to find the first answer



(b) Time to find all answers

Figure 9: Star graph execution time

Query keyword(s)	Answers	Answer trees	Time to 1st (ms)	Total time (ms)
Macron	118	0	179	390
Trump	10	0	26	36
Melenchon	8	0	31	39
Christophe, Dettinger	1105	319611	136	123932
Etienne, Chouard, Rodrigues	1	194	144	146
Thierry–Paul, Valette, Drouet	0	300813	N/A	120001
Melenchon, Aubry	9	284	38	929
Castaner, flashball	17	1724	61	545
Drouet, Levavasseur	18	518	145	309
Dupont–Aignan, Chalencon	21	1850	53	393
Estrosi, Castaner	16	2203	205	529
Alexis, Corbiere, Ruffin	11	3782	57	1022
Macron, Nunez	13	4107	1511	1561
Hamon, Drouet	5	421	71	145
Drouet, Ludosky	27	486	43	145
Salvini, Ludosky	17	1156	111	375
Salvini, Chouard	16	3205	76	710
Corbiere, Drouet	13	2341	129	673
Cauchy, Drouet	22	516	96	260
Benalla, Nunez	15	1027	199	347

Table 1: Results with real-world dataset

In [6], the authors investigate a different kind of answers to keyword search, the so-called r-clique graphs, which they solve with the help of specific indexes.

Keyword search across **heterogeneous datasets** has been previously studied in [11, 23]. However, in these works, *each answer comes from a single dataset*, that is, they never consider answers

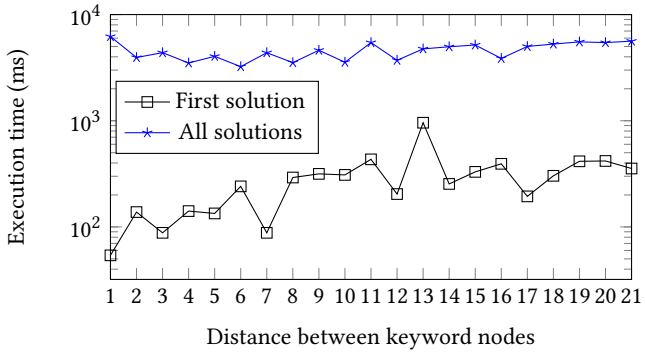


Figure 10: Barabasi-Albert graph execution time

spanning over and combining multiple datasets, such as the one shown in Figure 2.

In the literature, **(G)STP** has been addressed under various **simplifications** that do not hold in our context. For instance: the quality of a solution exponentially decreases with the tree size, thus search can stop when all trees are under a certain threshold [2]; edges are considered in a single direction [13, 22, 32]; the cost function has the suboptimal-structure property [10, 24] etc. These assumptions reduce the computational cost; in contrast, to leave our options open as to the best score function, we worked to build a feasible solution for the general problem we study. Some works have focused on finding **bounded (G)STP approximations**, i.e., (G)STP trees solutions whose cost is at most f times higher than the optimal cost, e.g., [15, 16]. Beyond the differences between our problem and (G)STP, due notably to the fact that our score is much more general (Section 3), non-expert users find it hard to set f.

Beyond the differences we mentioned above, most of which concern our bidirectional search, and the lack of favorable cost hypothesis, our work is the first to study querying of graphs originating from *integrating several data sources*, while at the same time *preserving the identity of each node from the original document*; this is a requirement for integrating, and simultaneously preserving, datasets of journalistic interest. In a companion paper [4] we present our latest algorithms for creating such graphs, relying also on information extraction, data matching, and named entity disambiguation; earlier versions were outlined in [5, 7].

Acknowledgements The authors would like to thank: Helena Galhardas and Julien Leblay who contributed to previous versions on this work [5, 7] and Tayeb Merabti for his support in the development and maintenance of the ConnectionLens system [4]. This work was partially supported by the H2020 research program under grant agreement nr. 800192, and by the ANR AI Chair SourcesSay.

REFERENCES

[1] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999). DOI: http://dx.doi.org/10.1126/science.286.5439.509
[2] Raphaël Bonaque, Bogdan Cautis, François Goasdoué, and Ioana Manolescu. 2016. Social, Structured and Semantic Search. In *EDBT*.
[3] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. 2015. Invisible Glue: Scalable Self-Tunning Multi-Stores. In *CIDR*.
[4] Oana Bălălău, Catarina Conceiąo, Helena Galhardas, Ioana Manolescu, Tayeb Merabti, Jingmao You, and Youssr Youssef. 2020. Graph integration of structured, semistructured and unstructured data for data journalism. BDA. (2020). https://hal.inria.fr/hal-02904797
[5] Camille Chanial, Rédouane Dziri, Helena Galhardas, Julien Leblay, Minh Huong Le Nguyen, and Ioana Manolescu. 2018. ConnectionLens: Finding Connections Across Heterogeneous Data Sources (demonstration). *VLDB* (2018).
[6] Yu-Rong Cheng, Ye Yuan, Jia-Yu Li, Lei Chen, and Guo-Ren Wang. 2016. Keyword Query over Error-Tolerant Knowledge Bases. *Journal of Computer Science and Technology* 31 (2016). Issue 4.
[7] Felipe Cordeiro, Helena Galhardas, Julien Leblay, Ioana Manolescu, and Tayeb Merabti. 2020. Keyword Search in Heterogeneous Data Sources. (2020). https://hal.inria.fr/hal-02559688 Technical report.
[8] Pericles de Oliveira, Altigran Soares da Silva, and Edleno Silva de Moura. 2015. Ranking Candidate Networks of relations to improve keyword search over relational databases. In *IEEE*.
[9] David J. DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. 2013. Split Query Processing in Polybase. In *SIGMOD*. DOI: http://dx.doi.org/10.1145/2463676.2463709
[10] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. 2007. Finding top-k min-cost connected trees in databases. In *ICDE*.
[11] Xin Dong and Alon Halevy. 2007. Indexing Dataspaces. In *SIGMOD*.
[12] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Rec.* 44, 2 (2015). DOI: http://dx.doi.org/10.1145/2814710.2814713
[13] Shady Elbassuoni and Roi Blanco. 2011. Keyword Search over RDF Graphs. In *CIKM*.
[14] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co. New York.
[15] N. Garg, G. Konjevod, and R. Ravi. 1998. A polylogarithmic approximation algorithm for the group Steiner tree problem. In *SIAM*.
[16] Andrey Gubichev and Thomas Neumann. 2012. Fast approximation of Steiner trees in large graphs. In *CIKM*.
[17] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. 2003. XRANK: Ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 16–27.
[18] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. 2007. BLINKS: ranked keyword searches on graphs. In *SIGMOD*.
[19] Vagelis Hristidis and Yannis Papakonstantinou. 2002. DISCOVER: Keyword Search in Relational Databases. In *VLDB*.
[20] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries over Heterogeneous Data through Engine Customization. *PVLDB* 9, 12 (2016). DOI: http://dx.doi.org/10.14778/2994509.2994516
[21] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. 2015. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*.
[22] Wangchao Le, Feifei Li, Anastasios Kementsietsidis, and Songyun Duan. 2014. Scalable Keyword Search on Large RDF Data. *IEEE Trans. Knowl. Data Eng.* 26, 11 (2014).
[23] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. 2008. EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured Data. In *SIGMOD*.
[24] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2016. Efficient and Progressive Group Steiner Tree Search. In *SIGMOD*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.).
[25] Ziyang Liu and Yi Chen. 2007. Identifying meaningful return information for XML keyword search. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 329–340.
[26] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-Foreign Language for Data Processing. In *SIGMOD (SIGMOD '08)*. DOI: http://dx.doi.org/10.1145/1376616.1376726
[27] Mayssam Sayyadian, Hieu LeKhac, AnHai Doan, and Luis Gravano. 2007. Efficient Keyword Search Across Heterogeneous Relational Databases. In *ICDE*.
[28] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-Reduce Framework. *PVLDB* 2, 2 (2009).
[29] Quang Hieu Vu, Beng Chin Ooi, Dimitris Papadias, and Anthony K. H. Tung. 2008. A graph method for keyword-based selection of the top-K databases. In *SIGMOD*.
[30] Zhepeng Yan, Nan Zheng, Zachary G. Ives, Partha Pratim Talukdar, and Cong Yu. 2015. Active learning in keyword search-based data integration. *VLDB J.* 24, 5 (2015).
[31] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. 2009. *Keyword Search in Databases*. DOI: http://dx.doi.org/10.2200/S00231ED1V01Y200912DTM001
[32] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. 2010. Keyword Search in Relational Databases: A Survey. *IEEE Data Eng. Bull.* 33, 1 (2010).
[33] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *CACM* 59, 11 (2016).